CEWES MSRC/PET TR/98-39

Performance Evaluation of HPF Kernels on the IBM SP and Cray T3E

by

Gina Goff
Charles Koelbel
Bob Robey
David Torres

07h00498

Work funded by the DoD High Performance Computing Modernization Program CEWES Major Shared Resource Center through

Programming Environment and Training (PET)

Supported by Contract Number: DAHC 94-96-C0002
Nichols Research Corporation

Performance Evaluation of HPF Kernels on the IBM SP and Cray T3E

Gina Goff, Charles Koelbel, Bob Robey, David Torres

June 12, 1998

1 Introduction

Migrating code to parallel machines can be a trying process. Unless users have good software and know how to exploit it, the promise of enhanced performance may seem more like a mirage than a readily attainable goal. Unfamiliarity with parallel languages and how their implementations are likely to behave can be a major obstacle for many users. To improve DOD users' knowledge in this area, we have executed several small tests representative of the sorts of operations commonly found in scientific applications on the SP and T3E at CEWES MSRC. The tests are written in High Performance Fortran (HPF), a data-parallel Fortran language available on both computers, and are a subset of benchmarks originally developed at the University of New Mexico by Bob Robey and Dave Torres. Their intent was to benchmark the performance of different HPF compilers on several computers to see how well HPF performed for a variety of implementations and platforms. The tests used for our research have been modified for CEWES MSRC machines, largely to compensate for the vagaries of floating-point implementations. Several calculations were changed to formulas having closed-form solutions to strengthen error checking. These tests are not meant to be absolute measures of the computers themselves, nor of the software written for them. Rather, our goal is to provide general guidelines for CEWES MSRC users by showing them which approaches are likely to yield the best performance in the environments already at their disposal. Our research was conducted as part of a benchmarking effort for HPF programs on the newer machines at CEWES MSRC. The results extend, however, to similar machines across DOD.

The tests fall into three categories: communication kernels, intrinsics, and computational kernels. Performance can be dependent on the idiosyncrasies of the compiler as well as the quirks of the hardware; thus, seemingly equivalent computations may behave quite differently. For that reason, multiple variants of many computations were tested, to see if undertaking an operation in a certain way had a significant impact on performance. Portland Group, Inc.'s HPF compiler was used and timings were taken under "normal" load conditions for both computers; i.e., the timings were not taken on dedicated machines. Each test was executed using 1, 4, 16, and 64 processors. The number of iterations that could be performed in approximately 300 seconds of CPU time was counted. For each test, the number of iterations performed was divided by the actual time spent in the critical loop to yield the number of iterations per second. (The time required to do array initialization, error checking, etc., was not included.) Speedup for 4, 16, and 64 processors was measured by dividing the iterations per second for multiple processors by the number of iterations per second for the uniprocessor case. We also compared the iterations per second for "optimized" loops to the iterations per second for the serial version on the same number of processors. The "uniprocessor" type of speedup indicates how well a particular test scales, while the "serial version" speedup is meant to show how effective a given optimization is. Test results will be discussed in detail in Sections 2 through 4, while some general conclusions will be presented in Section 5. Tables showing the speedups for all tests can be found in the appendix.

2 Communication Kernels

Four communication patterns were examined: all-to-all (all the nodes communicate with one another), gather/scatter (one node communicates with many nodes), reduction (determine minimum, maximum, sum, etc. of all nodes), and stencil (node communicates with its nearest neighbors).

2.1 All-to-All

2.1.1 Speedup Compared to Uniprocessor Case

Two common all-to-all operations, transpose and redistribute, were tested for a 320 by 320 element array. Table 1 shows the speedup for tests executed on multiple processors, compared to each loop's performance on a single processor. From left to right, Table 1 lists a number for each test, describes the critical loop for that test, and shows the speedup at 4, 16, and 64 processors for the SP and T3E.

For the transpose tests, the arrays were distributed (*,BLOCK) and several variations of the basic loop

```
do j = 1,ny
  do i = 1,nx
    y(i,j) = x(j,i)
  end do
end do
```

were exercised. In general, use of HPF's INDEPENDENT directive improved performance more than forall did, and transpose() and forall had about the same level of effectiveness. On the SP, cases where the blocked subscript position of the array being transposed varied with the outer loop index did better than those cases where the block-distributed subscript position was controlled by the inner loop, i.e.,

```
do i = 1,nx
  do j = 1,ny
    y(i,j) = x(j,i)
  end do
end do
```

performed better than

```
do j = 1,ny
  do i = 1,nx
    y(i,j) = x(j,i)
  end do
end do.
```

Both loops performed equally well on the T3E. The best performance on the SP was achieved by loops 6 and 7, both of them cases where the distributed subscript position varied with the outer loop and both loops were INDEPENDENT. Such tests also did well on the T3E, but INDEPENDENT loops where the distributed subscript position varied with the inner loop had slightly better performance.

For the redistribute tests, several methods were used to redistribute two-dimensional arrays, i.e.,

```
do j=1,ny
  do i=1,nx
    z2(i,j)=z1(i,j)
  end do
end do
```

where z1 is (*,BLOCK) and z2 is (BLOCK,*). Tests where the distributed subscript position varied with the outer loop index and both loops were INDEPENDENT had the best speedup for both the SP and T3E. Array syntax (e.g., z2=z1) and forall had roughly equivalent speedups for both machines. The REDISTRIBUTE directive performed slightly better than forall on the SP, but slightly worse on the T3E.

2.1.2 Speedup Compared to Serial Version

Table 2 shows the speedup for each optimized loop, compared to the performance of its serial version. For the SP, transpose() had the best speedup, followed by forall. The speedup dropped as the number of processors increased for INDEPENDENT, but not for transpose() or forall.

For the transpose tests run on the T3E, INDEPENDENT had better speedup than forall. The transpose() intrinsic and forall had about the same performance, except for loop 9 (which is the basic serial version shown above plus forall), where the speedup dropped sharply as the number of processors increased.

For the SP, REDISTRIBUTE had the best speedup, with (BLOCK,*) doing slightly better than (*,BLOCK). INDEPENDENT did well at 64 processors, but had erratic behavior for smaller numbers of processors.

REDISTRIBUTE also had excellent speedup on the T3E, but array syntax and forall did just as well. INDEPENDENT was "worst", but still provided a speedup of 2000 for 64 processors.

2.2 Gather/Scatter

2.2.1 Speedup Compared to Uniprocessor Case

The basic gather/scatter tests were executed using a 442-element block-distributed source array and a 24 by 24 by 24 target array that was distributed (*,*,BLOCK). The dense gather/scatter tests used a block-distributed 3143-element array. Table 3 shows the speedup for tests run on multiple processors, compared to each test's performance on a single processor. For the basic scatter loop,

```
do i=1,n
  u(f(i,1),f(i,2),f(i,3)) = v(i)
end do
```

COPY_SCATTER had the largest speedup, and INDEPENDENT did no better than the serial loop on both machines. Forall did better than INDEPENDENT on the T3E; the opposite was true for the SP.

For the dense scatter loop,

```
do i=1,n
  ug1(i) = ug2(ds(i))
end do
```

INDEPENDENT had the most speedup on the SP, followed by the serial version of the loop. On the T3E, forall performed best, with the serial version coming in second and the INDEPENDENT loop a distant third.

The basic gather test,

```
do i=1,n
  v(i) = u(f(i,1),f(i,2),f(i,3))
end do
```

had the most speedup with forall (loop 8), followed by the serial version, on both machines. When a temporary variable was used (loop 9), forall fell behind the serial version.

The dense gather,

```
do i=1,n
  ug2(ds(i)) = ug1(i)
end do
```

had results similar to the dense scatter test. INDEPENDENT did better than forall on the SP, with the serial version coming in third. On the T3E, forall had the most speedup, followed by the serial version.

2.2.2 Speedup Compared to Serial Version

Table 4 shows the speedup for each optimized loop, compared to the performance of its serial equivalent. For the basic scatter operation, forall had the most speedup on both machines, with COPY_SCATTER coming in second. The speedups for both loops show a dip at 16 processors on the SP, while the speedups for the T3E are monotonically increasing. Speedups for UNPACK dropped as the number of processors increased, for both machines. Both the dense scatter and dense gather tests showed greater speedups for INDEPENDENT than for

forall, especially on the T3E.

For the basic gather operation, the SP and T3E had similar results. The forall test without the temporary variable (loop 8) had larger speedups than the forall test with the temporary, and both loops did better than INDEPENDENT. Both PACK and the forall with the temporary variable had speedups that decreased as the number of processors increased.

2.3 Reduction

2.3.1 Speedup Compared to Uniprocessor Case

Two common reduction operations, maximum and sum, were tested using block-distributed 1000-element arrays. Table 5 shows the speedup for test loops executed on multiple processors, compared to each test's performance on a single processor.

For maximum, we began with the simple loop

```
do i=1,n
  r=max(x(i),r)
end do
```

and tried several variations. Using INDEPENDENT with REDUCTION yielded the best speedup for both machines. On the SP, forall was second-best, followed closely by maxval; maxval did slightly better than forall on the T3E. Speedup decreased as the number of processors increased for all of the loops.

For sum reduction, i.e.,

```
do i=1,n
  r=r+x(i)
end do
```

sum() had the best speedup on the SP, closely followed by the combination of INDEPENDENT and REDUCTION. On the T3E, INDEPENDENT plus REDUCTION did the best, while sum() and INDEPENDENT alone were both slightly behind the performance of the basic loop.

When a more complicated sum reduction was considered,

```
do i=1,n                                        (loop 17)
  w(i)=r+abs(x(i)-u(i))
end do
rsum=sum(w)
```

where u and w are also block-distributed, the results were somewhat different. Loop 17, as shown above, had the greatest speedup on the SP, followed by loop 18 (loop 17 plus INDEPENDENT). For the T3E, the order was reversed. The test

```
rsum=sum(abs(x-u))
```

came in third on the T3E, but was seventh on the SP. For both machines, all the other tests using array syntax and the test using forall had roughly the same amount of speedup. Again, speedup decreased as the number of processors increased for all tests.

2.3.2 Speedup Compared to Serial Version

Table 6 shows the speedup for each optimized loop, compared to the performance of its serial version. For the basic "maximum" reduction loop,

```
do i=1,n
  r=max(x(i),r)
end do
```

maxval had the best speedup on both machines, followed by INDEPENDENT plus REDUCTION. Speedups for INDEPENDENT + REDUCTION improved as the number of processors increased on the T3E; the opposite was true for the SP. When we changed the loop slightly to read

```
do i=1,n
  r=max(abs(x(i)-u(i)),r)
end do
```

forall had the largest speedups for both machines. For the SP, maxval and INDEPENDENT with REDUCTION came in second and third; on the T3E, the order was reversed.

For the simple sum reduction test, sum() had the best speedups on both machines, followed by the INDEPENDENT plus REDUCTION combination. For the more complex

```
do i=1,n
  w(i)=r+abs(x(i)-u(i))
end do
rsum=sum(w)
```

most of the variations had the same amount of speedup on both machines, with the exception of INDEPENDENT (without REDUCTION) and sum(abs(x-u)), which lagged behind.

2.4 Stencil

2.4.1 Speedup Compared to Uniprocessor Case

We performed a simple 5-point stencil computation on a 1000 by 1000 array:

```
do j=1,n
  do i=1,n
    UN(i,j) = [U(i,j) + U(i+1,j) + U(i-1,j) + U(i,j+1) + U(i,j-1)]/5
  end do
end do
```

Table 7 shows the speedup for tests run on multiple processors, compared to each test's speedup on one processor. Loop 4, where both arrays were distributed (*,BLOCK) and forall and eoshift were used, had the greatest speedup on both machines.

On the SP, tests with forall had only slightly larger speedups than tests using INDEPENDENT, regardless of the distribution. Tests where the arrays were distributed (BLOCK,BLOCK) had less speedup than the ones using a (*,BLOCK) distribution.

On the T3E, forall's performance was slightly less than INDEPENDENT's when the arrays were distributed (*,BLOCK), but the gap was much larger when a (BLOCK,BLOCK) distribution was used. One of the (BLOCK,BLOCK) tests (loop 5) had speedups similar to the ones seen for the (*,BLOCK) tests, but the other (BLOCK,BLOCK) loops did not fare so well.

2.4.2 Speedup Compared to Serial Version

Table 8 shows the speedups for tests, compared to the performance of the serial version. On the SP, loops 2 and 3 (arrays distributed (*,BLOCK); either INDEPENDENT or forall used) had the best speedups. The (BLOCK,BLOCK) loops were next, and had similar speedups. Loop 4, which had the largest speedup relative to the uniprocessor case, had the lowest speedup here.

For the T3E, loop 3 (arrays distributed (*,BLOCK); both loops INDEPENDENT) easily had the largest amount of speedup. Loops 6 and 7 were next, but not all the (BLOCK,BLOCK) loops had good speedups: the two forall tests (6 and 7) were fairly close to each other, but loop 5 (i and j loops INDEPENDENT) lagged far behind. Unlike the SP, where loops 2 and 3 had equivalent speedups, forall did much better than

INDEPENDENT.

3 Intrinsic Kernels

Two intrinsic functions, transpose and redistribute, were tested on 32 by 32 by 32 arrays. For the transpose tests, arrays were always distributed as (*,*,BLOCK). The redistribute tests changed the distribution from (*,*,BLOCK) to (*,BLOCK,*). The speedups for both kinds of tests can be found in Tables 9 and 10.

3.1 Transpose

3.1.1 Speedup Compared to Uniprocessor Case

For the transpose tests, the basic loop was

```
do k=1,nz
  do j=1,ny
    do i=1,nx
      y(i,j,k) = x(i,k,j)
    end do
  end do
end do
```

For both machines, the best speedup was obtained using colon notation for two dimensions, i.e.,

```
do i=1,nx
  y(i,:,:) = transpose(x(i,:,:))
end do
```

On the SP, the use of colon notation in one dimension came in second, followed by the serial version; the opposite was true on the T3E. For both machines, speedups decreased as the number of processors increased.

3.1.2 Speedup Compared to Serial Version

On both machines, loop 3, which used colon notation in two dimensions, had larger speedups than the use of colon notation in only one dimension (loop 2). The amount of speedup decreased as the number of processors increased, for loop 2 on both machines and for loop 3 on the SP. For the T3E, speedup increased with the number of processors for loop 3.

3.2 Redistribute

3.2.1 Speedup Compared to Uniprocessor Case

The basic loop was

```
do k=1,nz
  do j=1,ny
    do i=1,nx
      x2(i,j,k) = x(i,j,k)
    end do
  end do
end do
```

On the SP, array syntax (x2 = x) and REDISTRIBUTE (*,BLOCK,*) had equivalent speedups. Array syntax did better than REDISTRIBUTE on the T3E. As the number of processors increased, speedups decreased on the SP and increased on the T3E.

3.2.2 Speedup Compared to Serial Version

For both machines, loop 5, which used array syntax, had larger speedups than loop 6, which used REDISTRIBUTE, although the differences were not huge. Speedups increased as the number of processors increased for both loops.

4 Computational Kernels

4.1 Matrix Addition

4.1.1 Speedup Compared to Uniprocessor Case

Matrix addition was tested for two-dimensional 1000-element arrays. The speedups for both kinds of tests can be found in Tables 11 and 12. The basic loop was

```
do j=1,n
  do i=1,n
    C(i,j) = A(i,j) + B(i,j)
  end do
end do
```

Speedup for the serial loop dropped slightly as the number of processors increased on the SP, but remained constant for the T3E. When the arrays were distributed (*,BLOCK) and array syntax was used (C = A + B), speedups were considerably larger for the SP than for the T3E.

4.1.2 Speedup Compared to Serial Version

Although the SP had better speedups compared to the single-processor case, the T3E shows more improvement over the serial version of the loop. Thus, the loop scales better for the SP, but is a better optimization for the T3E.

4.2 Matrix Multiplication

4.2.1 Speedup Compared to Uniprocessor Case

Y = A * X was tested for 500 by 500 matrices, using three different algorithmic approaches: inner product oriented, column-oriented, and row-oriented. Table 13 shows the speedup for tests executed on multiple processors versus the single-processor case, for all three approaches.

Inner Product

For the multiplication:

```
do i=1,n
  do j=1,n
    do k=1,m
      Y(i,j) = Y(i,j) + A(i,k) * X(k,j)
    end do
  end do
end do
```

where m = n = 500, the SP and T3E had the greatest amount of speedup for loop 5, where the two outermost loops were INDEPENDENT and all three arrays were distributed (BLOCK,BLOCK).

For the SP, the use of INDEPENDENT alone had similar performance to loop 2, where X and Y were distributed (BLOCK,*) and A was distributed (*,BLOCK.

On the T3E, loop 4 (A and Y distributed (BLOCK,*); X distributed (*,BLOCK); i and j loops INDEPENDENT) was second-best, followed by the use of INDEPENDENT alone.

For the multiplication:

```
do i=1,n
  do j=1,m
    do k=1,n
      Y(k,i) = Y(k,i) + A(k,j) * X(j,i)
    end do
  end do
end do
```

loop 5 (A, X, Y distributed (BLOCK,BLOCK); i and j loops INDEPENDENT) was again the best performer on both the SP and T3E.

For the SP, loop 2 (all arrays distributed (*,BLOCK)) was slightly better than loop 5 for 4 and 16 processors, but not for 64. Loop 4, where the i and k loops were INDEPENDENT and all three arrays were distributed, did no better than loop 3, where i and k were INDEPENDENT and none of the arrays were distributed.

On the T3E, loop 3 had larger speedups than loop 4, and the disparity increased with the number of processors.

Row-oriented

For the multiplication:

```
do i=1,n
  do j=1,m
    do k=1,n
      Y(i,k) = Y(i,k) + A(i,j) * X(j,k)
    end do
  end do
end do
```

loop 5 (all arrays distributed (BLOCK,BLOCK); i and k loops INDEPENDENT) had the greatest speedups on both machines. This was especially true for the SP, where the speedup for loop 5 dwarfs all the other tests.

For both machines, loop 4 (i and k loops INDEPENDENT; A distributed (*,BLOCK); X and Y distributed (BLOCK,*)) came in second. Loop 4 was only slightly better than loop 3, where the i and k loops were INDEPENDENT but none of the arrays were distributed.

4.2.2 Speedup Compared to Serial Version

Inner Product

Table 14 shows the speedup for each "optimized loop", compared to the performance of its serial counterpart. For the SP, loop 5 (all arrays distributed (BLOCK,BLOCK); outermost two loops INDEPENDENT) had the greatest speedup, followed by INDEPENDENT alone and loop 2 (X,Y are (BLOCK,*); A is (BLOCK)).

On the T3E, loop 2 had the best speedup, followed by loop 5 and then loop 4 (i and j loops are INDEPENDENT; X is (*,BLOCK); A,Y are (BLOCK,*)). The use of INDEPENDENT alone achieved speedups nearly as good as those for loop 4, where the arrays were distributed.

Column-oriented

Both the SP and T3E had results similar to those for the inner product approach: for each, the best and second-best speedups were for the same loops as before. On the SP, loop 4 was in third place and only slightly behind loop 3, while loop 2 (all arrays are (*,BLOCK)), which did well in the inner product approach, had abysmal performance for this test.

On the T3E, loop 3 (i and k loops are INDEPENDENT) was third, only slightly ahead of loop 4, which was third for the inner product case.

Row-oriented

Loop 5 (A,X,Y distributed (BLOCK,BLOCK); i and k loops INDEPENDENT) had the most speedup for both the SP and T3E. For the SP, loop 2 (all arrays distributed (BLOCK,*)) was second-best, followed by loops 3 and 4, which had equivalent performance.

For the T3E, loop 4 (X,Y are (BLOCK,*); A is (*,BLOCK); i and k loops INDEPENDENT)) had the second largest speedup, and loop 3 (i and k loops INDEPENDENT) did nearly as well.

4.3 Multiply/Add by Matrix Element

4.3.1 Speedup Compared to Uniprocessor Case

Element-by-element matrix multiplication/addition was tested on 1000 by 1000 arrays. Arrays were distributed (*,BLOCK) for all tests using distributed arrays. Four serial loops were tested, along with one optimization for each serial version. Speedups are shown in Tables 15 and 16.

Loop 1 was a straightforward matrix multiply:

```
do j=1,n
  do i=1,n
    C(i,j) = A(i,j) * B(i,j)
  end do
end do
```

For loop 2, all arrays were distributed and array syntax was used. Speedups for loop 2 steadily increased on the SP, but not on the T3E, where the speedup dropped sharply at 16 processors.

In a slightly more complex loop,

```
C(i,j) = A(i,j) * B(i,j)
```

was changed to

```
C(i,j) = A(i,j) * B(i,j) + D(i,j)
```

Again, the SP had better speedups, both for the serial version and for the optimization, where all arrays were distributed and array syntax was used. This time, however, the T3E's speedup for the optimized loop peaked at 16 processors and then dropped off.

The next test, loop 5, tested multiplication by a scalar constant, i.e.,

```
do j=1,n
  do i=1,n
    C(i,j) = A(i,j) * k
  end do
end do
```

For the SP, the optimized version (A, C distributed; array syntax used) had much better speedups than the serial

loop for 4 and 16 processors, but the advantage was slightly less at 64 processors. On the T3E, the speedup was the same for 4, 16, and 64 processors for loop 5, but peaked at 16 processors for the optimized version.

Loop 7 tested multiplication by and addition with a scalar constant, i.e.,

```
do j=1,n
  do i=1,n
    C(i,j) = A(i,j) * k + m
  end do
end do
```

The serial version had a constant speedup on both machines. The SP showed less improvement in speedup for the optimized version (loop 8) than it did for the optimized variation of loop 5. On the T3E, as before, the speedup peaked at 16 processors and was not very promising overall.

4.3.2 Speedup Compared to Serial Version

Loop 2,

```
do j=1,n
  do i=1,n
    C = A * B
  end do
end do
```

where all arrays were distributed, had the best speedup on both machines. Speedups for the other three optimized loops dropped off at 64 processors for the T3E; on the SP, this only happened for loop 6 (C = A * k).

5 Conclusions

In general, communications were expensive, especially reduction, but this poor performance was typically offset by good serial version speedups. Intrinsics also had mediocre uniprocessor-type speedup, but showed excellent serial version speedups that increased with the number of processors, most notably when array syntax or REDISTRIBUTE was used. It should be remembered that the test arrays we used were relatively small and that the uniprocessor performance should be better for much larger arrays.

The use of array syntax (a = b instead of a(i) = b(j), where a and b are arrays) was usually advantageous. (BLOCK,BLOCK) distributions did extremely well for matrix multiplication but had uneven performance for the stencil computations, probably because of cache performance. Cases where (*,BLOCK) distributions were used with INDEPENDENT did better than cases where INDEPENDENT was used alone, but not by much. For (*,BLOCK) or (BLOCK,*) distributions on the SP, tests where the distributed subscript position of the input array varied with the outer loop index often did better than those cases where the subscript position was controlled by the inner loop. Both cases performed equally well on the T3E.

An important aspect of HPF is its promise of a "efficiently portable programming style". That is, when there are several ways to express the same computation, the relative efficiencies of different variants should not vary wildly from machine to machine. We tested this aspect of HPF as follows: for any test that could be expressed in more than one way, we ranked the variations according to their speedups (over single-processor execution) for each machine and then counted how many variants were ranked in the top two or three on both computers. Table 17 shows how many loops were ranked as being in the top two or three on both machines, and also how many loops were ranked in the same order on both machines. Figure 1, which summarizes Table 17, indicates how many tests were top performers on both the SP and T3E. For example, the first set of columns in Figure 1 shows that for 16 tests, only one variant placed in the top two on both machines, while there were 12 tests that had the same variants ranked in the top two. The second set of columns shows similar results when comparing the top three performers. The third set of columns indicates how many variants appeared in the top rankings in the same order, i.e., there were 10 cases where only the first-place finisher was the same, but 5 cases where both machines

had the same variants in first, second, and third place. This implies that programs that scale well on one computer will probably have acceptable if not optimal behavior on the other**.**

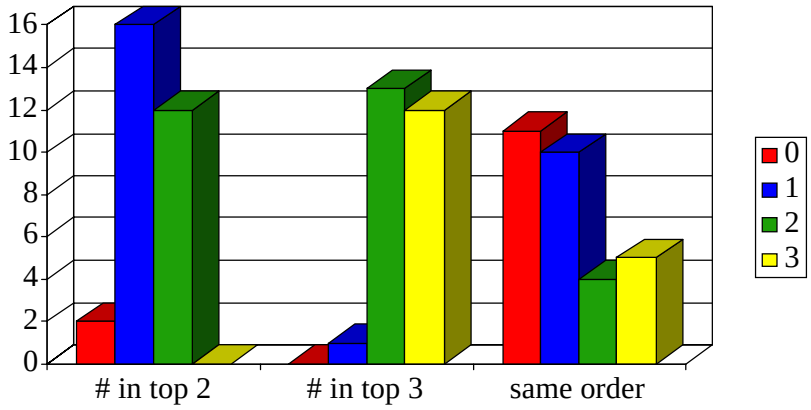

Figure 1 — # of Loops Ranked in Top 2 or 3 on Both Machines

Speedup compared to uniprocessor performance indicates how well a test scales, while speedup compared to the serial version's performance indicates how much benefit is derived from that particular optimization. Ideally, we would like the constructs that scale the best to also optimize well. Figure 2, which summarizes Table 18, addresses the question of how many loops performed well according to both of our definitions of speedup. For each machine, we ranked the variants twice: once according to uniprocessor speedup and once according to serial version speedup. We then compared the rankings to determine whether the same variants were ranked highly for both kinds of speedup. Figure 2 shows two sets of columns for each machine. The set on the left indicates how many variants were ranked in the top two for both kinds of speedup, while the set on the right shows how many variants were ranked in the top three. For example, on the SP, there were eight cases where two out of the three highest-ranked variants for both definitions of speedup were the same and two cases where all three of the highest-ranked variants were the same. Figure 2 shows that many of the best loops had both kinds of speedup, indicating that variants that scale well also tend to be good optimizations.

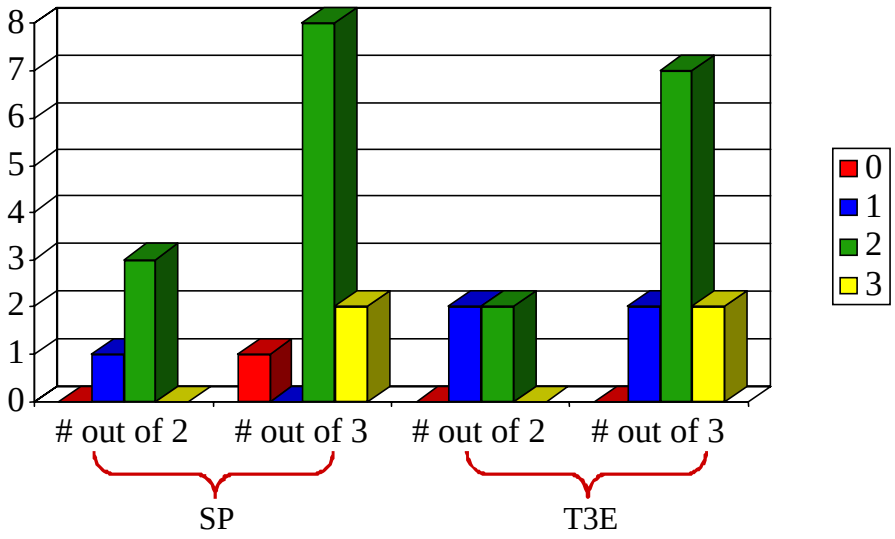

Figure 2 — # of Loops Ranked Highly for Both Kinds of Speedup

6 Acknowledgements

This work was supported in part by a grant of HPC time from the DoD HPC Modernization Program.

Appendix A: Detailed Test Results

		SP			T3E		
	number of processors:	**4**	**16**	**64**	**4**	**16**	**64**
Transpose for (*,BLOCK) distributions							

1	do j=1,ny
	  do i=1,nx
	  y(i,j)=x(j,i)
		0.1	0.4	0.1	0.8	0.5	0.2

2	do i=1,nx
	  do j=1,ny
	  y(i,j)=x(j,i)
		0.6	0.4	0.1	0.8	0.5	0.2

3	do j=1,ny
	  do i=1,nx
	  y(j,i)=x(i,j)
		0.6	0.4	0.1	0.8	0.5	0.2

4	do i=1,nx
	  do j=1,ny
	  y(j,i)=x(i,j)
		0.1	0.4	0.1	0.8	0.5	0.2

5	test 1; both loops INDEPENDENT	2.4	5.1	0.2	4.5	12.7	7.5
6	test 2; both loops INDEPENDENT	3.9	7.8	2.8	4.5	13.6	27.7
7	test 3; both loops INDEPENDENT	3.9	7.8	1.7	4.6	13.8	28.1
8	test 4; both loops INDEPENDENT	2.6	5.0	0.1	4.6	14.5	28.1
9	test 1, with forall	2.5	4.8	2.7	3.9	6.0	0.6
10	test 2, with forall	2.5	4.9	3.7	3.9	7.3	10.8
11	test 3, with forall	2.5	4.9	3.7	3.8	7.2	10.7
12	test 4, with forall	2.5	4.8	3.1	3.8	7.2	10.8
13	y=transpose(x)	2.5	4.8	3.7	3.9	7.3	10.8
Redistribute from (*,BLOCK) to (BLOCK,*)							

14	do j=1,ny
	  do i=1,nx
	  z2(i,j)=z1(i,j)
		0.6	0.4	0.1	0.8	0.5	0.2

15	do i=1,nx
	  do j=1,ny
	  z2(i,j)=z1(i,j)
		0.1	0.4	0.1	0.8	0.5	0.2

16	test 14; both loops INDEPENDENT	0.1	1.5	1.3	3.7	8.8	12.7
17	test 15; both loops INDEPENDENT	3.5	8.4	9.8	4.2	14.2	19.1
18	test 14, with forall	0.5	0.9	0.6	4.1	5.7	5.2
19	test 15, with forall	0.5	0.9	0.5	4.0	5.7	5.2
20	z2=z1	0.5	0.9	0.5	4.1	5.6	5.2
21	!HPF$ REDISTRIBUTE z3(BLOCK,*)	0.5	0.9	0.7	3.4	4.3	4.4
22	!HPF$ REDISTRIBUTE z3(*,BLOCK)	0.5	1.0	0.7	3.2	4.4	3.8

Table 1 — All-to-All, Speedup Compared to Uniprocessor Case

		SP				T3E			
	number of processors:	**1**	**4**	**16**	**64**	**1**	**4**	**16**	**64**
Transpose for *,BLOCK distributions									

1 - serial version	do j=1,ny
	  do i=1,nx
	  y(i,j)=x(j,i)								

5 vs. 1	loops INDEPENDENT	62	1565	889	76	27	153	702	970
9 vs. 1	with forall	72	1913	986	1573	62	304	767	179
13 vs. 1	using transpose()	72	1913	986	2132	62	303	941	3233

2 - serial version	do i=1,nx
	  do j=1,ny
	  y(i,j)=x(j,i)								

6 vs. 2	loops INDEPENDENT	44	292	849	935	28	165	824	3914
10 vs. 2	with forall	76	324	928	2112	60	304	948	3262

3 - serial version	do j=1,ny
	  do i=1,nx
	  y(j,i)=x(i,j)								

7 vs. 3	loops INDEPENDENT	41	275	837	569	27	165	817	3909
11 vs. 3	with forall	70	312	914	2110	60	299	923	3258

4 - serial version	do i=1,nx
	  do j=1,ny
	  y(j,i)=x(i,j)								

8 vs. 4	loops INDEPENDENT	59	1645	840	36	26	158	808	3678
12 vs. 4	with forall	70	1864	945	1780	60	302	928	3269
Redistribute from *,BLOCK to BLOCK,*									

14 - serial version	do j=1,ny
	  do i=1,nx
	  z2(i,j)=z1(i,j)								

16 vs. 14	loops INDEPENDENT	194	29	775	2118	36	169	659	2255
18 vs. 14	with forall	385	313	916	1824	153	796	1832	3909
20 vs. 14	z2=z1	385	320	916	1683	151	796	1766	3909
21 vs. 14	REDISTRIBUTE z3(BLOCK,*)	392	335	933	2118	180	783	1596	3909
22 vs. 14	REDISTRIBUTE z3(*,BLOCK)	349	335	933	1876	194	796	1766	3665

15 - serial version	do i=1,nx
	  do j=1,ny
	  z2(i,j)=z1(i,j)								

17 vs. 15	loops INDEPENDENT	28	988	640	2137	21	117	640	2017
19 vs. 15	with forall	421	1890	977	1743	153	794	1850	3900

Table 2 — All-to-All, Speedup Compared to Serial Version

		SP			T3E		
	number of processors:	**4**	**16**	**64**	**4**	**16**	**64**
scatter							
1	do i=1,n u(f(i,1),f(i,2),f(i,3))=v(i)	0.95	0.99	0.63	0.80	0.85	0.42
2	with INDEPENDENT	0.97	0.97	0.64	0.82	0.87	0.43
3	with forall	0.79	0.71	0.46	1.00	1.23	0.81
4	u=UNPACK(v,MASK= maskf, FIELD=(0,0))	0.07	0.04	0.03	0.07	0.04	0.03
5	u=COPY_SCATTER (v,zero_u,f(:,1),f(:,2) f(:,3),truev)	1.32	0.85	0.86	1.60	2.20	1.55
gather							
6	do i=1,n v(i)=u(f(i,1),f(i,2),f(i,3))	0.75	0.84	0.38	0.81	0.86	0.42
7	with INDEPENDENT	0.12	0.06	0.02	0.49	0.15	0.03
8	forall(i=1:n) v(i)=u(f(i,1),f(i,2),f(i,3))	0.77	0.63	0.33	0.95	1.18	0.80
9	forall(i=1:n) t(i)=u(f(i,1),f(i,2),f(i,3)) v=t	0.51	0.30	0.12	0.58	0.42	0.22
10	v=PACK(u,MASK= maskf)	0.06	0.03	0.02	0.06	0.03	0.02
dense scatter							
11	do i=1,n ug1(i) = ug2(ds(i))	0.14	0.64	0.98	0.75	0.55	0.26
12	with INDEPENDENT	1.26	1.14	0.92	0.53	0.35	0.18
13	with forall	0.12	0.11	0.10	1.46	2.33	2.60
dense gather							
14	do i=1,n ug2(ds(i)) = ug1(i)	0.001	0.002	0.002	0.76	0.55	0.26
15	with INDEPENDENT	1.26	1.16	1.15	0.57	0.39	0.20
16	with forall	1.08	0.98	0.93	1.49	2.40	2.70

Table 3 — Gather/Scatter, Speedup Compared to Uniprocessor Case

		SP				T3E			
	number of processors:	**1**	**4**	**16**	**64**	**1**	**4**	**16**	**64**
scatter									
1 - serial version	do i=1,n u(f(i,1),f(i,2),f(i,3))=v(i)								
2 vs. 1	with INDEPENDENT	1	1	1	1	1	1	1	1
3 vs. 1	with forall	16	13	11	12	22	28	32	43
4 vs. 1	u=UNPACK(v,MASK= maskf, FIELD=(0,0))	0.4	0.03	0.02	0.02	0.94	0.09	0.05	0.07
5 vs. 1	u=COPY_SCATTER (v,zero_u,f(:,1),f(:,2) f(:,3),truev)	7	10	6	10	9	18	23	33
gather									
6 - serial version	do i=1,n v(i)=u(f(i,1),f(i,2),f(i,3))								
7 vs. 6	with INDEPENDENT	11	2	0.8	0.7	11	7	2	1
8 vs. 6	forall(i=1:n) v(i)=u(f(i,1),f(i,2),f(i,3))	16	16	12	14	22	26	30	42
9 vs. 6	forall(i=1:n) t(i)=u(f(i,1),f(i,2),f(i,3)) v=t	13	9	5	4	22	16	11	12
10 vs. 6	v=PACK(u,MASK= maskf)	0.6	0.04	0.02	0.03	1	0.09	0.05	0.07
dense scatter									
11 - serial version	do i=1,n ug1(i) = ug2(ds(i))								
12 vs. 11	with INDEPENDENT	20	178	35	18	0.9	0.7	0.6	0.6
13 vs. 11	with forall	218	193	38	23	35	67	147	343
dense gather									
14 - serial version	do i=1,n ug2(ds(i)) = ug1(i)								
15 vs. 14	with INDEPENDENT	0.02	21	13	13	0.4	0.3	0.3	0.3
16 vs. 14	with forall	0.05	43	26	25	34	67	151	352

Table 4 — Gather/Scatter, Speedup Compared to Serial Version

		SP			T3E		
	number of processors:	**4**	**16**	**64**	**4**	**16**	**64**
1	do i=1,n r=max(x(i),r)	0.04	0.02	0.01	0.06	0.03	0.02
2	test 1, with INDEPENDENT	0.04	0.02	0.01	0.06	0.04	0.03
3	test 1, with INDEPENDENT and REDUCTION	0.27	0.11	0.05	1.40	1.22	0.97
4	r=maxval(x)	0.23	0.13	0.06	0.57	0.39	0.30
5	do i=1,n r=max(abs(x(i)-u(i),r)	0.04	0.02	0.01	0.05	0.03	0.02
6	test 5, with INDEPENDENT	0.04	0.02	0.01	0.05	0.03	0.02
7	test 5, with INDEPENDENT and REDUCTION	0.30	0.17	0.07	1.42	1.28	1.02
8	test 5, with forall	0.26	0.13	0.10	0.85	0.66	0.51
9	test 5, with maxval	0.21	0.14	0.06	0.89	0.70	0.60
10	do i=1,n r=r+x(i)	0.03	0.02	0.01	0.06	0.04	0.03
11	test 10, with INDEPENDENT	0.03	0.02	0.01	0.05	0.03	0.02
12	test 10, with INDEPENDENT and REDUCTION	0.06	0.03	0.01	1.32	1.16	0.91
13	r=sum(x)	0.07	0.03	0.02	0.44	0.30	0.22
14	do i=1,n r=r+abs(x(i)-u(i))	0.04	0.02	0.01	0.05	0.03	0.02
15	test 14, with INDEPENDENT	0.03	0.02	0.01	0.05	0.03	0.02
16	test 14, with INDEPENDENT and REDUCTION	0.12	0.05	0.03	1.37	1.23	0.97
17	do i=1,n w(i)=r+abs(x(i)-u(i)) end do rsum=sum(w)	0.26	0.15	0.08	0.96	0.90	0.84
18	test 17, with INDEPENDENT	0.21	0.11	0.06	1.48	1.37	1.10
19	test 17, with forall	0.17	0.09	0.05	0.70	0.53	0.41
20	w(:)=abs(x(:)-u(:)) rsum=sum(w(:))	0.17	0.09	0.05	0.69	0.52	0.40
21	w(:)=abs(x(:)-u(:)) rsum=sum(w)	0.16	0.09	0.05	0.67	0.50	0.39
22	w=abs(x-u) rsum=sum(w)	0.17	0.09	0.06	0.68	0.51	0.39
23	rsum=sum(abs(x-u))	0.04	0.07	0.04	0.81	0.63	0.53

Table 5 — Reduction, Speedup Compared to Uniprocessor Case

		SP				T3E			
	number of processors:	**1**	**4**	**16**	**64**	**1**	**4**	**16**	**64**
1 - serial version	do i=1,n r=max(x(i),r)								
2 vs. 1	with INDEPENDENT	1	1	1	1	1	1	1	1
3 vs. 1	with INDEPENDENT and REDUCTION	11	83	62	41	3	80	119	134
4 vs. 1	with maxval	13	85	81	62	11	106	125	137
5 - serial version	do i=1,n r=max(abs(x(i)-u(i),r)								
6 vs. 5	with INDEPENDENT	1	1	1	1	1	1	1	1
7 vs. 5	with INDEPENDENT and REDUCTION	20	157	155	92	5	151	233	261
8 vs. 5	with forall	20	135	117	147	11	181	241	263
9 vs. 5	with maxval	23	127	152	103	8	144	193	235
10 - serial version	do i=1,n r=r+x(i)								
11 vs. 10	with INDEPENDENT	1	1	1	1	1	1	1	1
12 vs. 10	with INDEPENDENT and REDUCTION	36	80	79	51	4	81	121	133
13 vs. 10	r=sum(x)	36	87	80	73	16	110	126	135
14 - serial version	do i=1,n r=r+abs(x(i)-u(i))								
15 vs. 14	with INDEPENDENT	1	1	1	1	1	1	1	1
16 vs. 14	with INDEPENDENT and REDUCTION	40	139	95	87	5	149	230	256
17 - serial version	do i=1,n w(i)=r+abs(x(i)-u(i)) end do rsum=sum(w)								
18 vs. 17	with INDEPENDENT	1	1	1	1	2	3	3	3
19 vs. 17	with forall	2	1	1	1	6	4	4	3
20 vs. 17	w(:)=abs(x(:)-u(:)) rsum=sum(w(:))	2	1	1	1	6	5	4	3
21 vs. 17	w(:)=abs(x(:)-u(:)) rsum=sum(w)	2	1	1	1	7	5	4	3
22 vs. 17	w=abs(x-u) rsum=sum(w)	2	1	1	1	6	5	4	3
23 vs. 17	rsum=sum(abs(x-u))	2	0.3	1	1	4	3	3	3

Table 6 — Reduction, Speedup Compared to Serial Version

		SP			T3E		
	number of processors:	**4**	**16**	**64**	**4**	**16**	**64**

1	do j=1,n
	  do i=1,n
	  UN(i,j) = (U(i,j) +
	  U(i+1,j) + U(i-1,j) +
	  U(i,j+1) + U(i,j-1))/5
		1.0	1.0	0.9	1.2	1.2	1.2

2	U, UN are (*,BLOCK); i,j loops INDEPENDENT	3.9	13.9	34.4	5.5	16.0	40.9
3	U, UN are (*,BLOCK); with forall	3.4	13.8	34.3	4.1	15.2	40.0
4	U, UN are (*,BLOCK); with forall and eoshift	3.8	14.1	35.6	3.6	15.0	43.6
5	U, UN (BLOCK,BLOCK); i,j loops INDEPENDENT	3.5	11.6	27.7	4.2	15.5	41.1
6	U, UN (BLOCK,BLOCK); with forall	3.6	11.6	27.9	3.6	10.6	27.7
7	U, UN (BLOCK,BLOCK); with forall and eoshift	3.6	11.8	28.3	3.6	10.9	28.3

Table 7 — 5-Point Stencil, Speedup Compared to Uniprocessor Case

		SP				T3E			
	number of processors:	**1**	**4**	**16**	**64**	**1**	**4**	**16**	**64**

1 - serial version	do j=1,n
	  do i=1,n
	  UN(i,j) = (U(i,j) +
	  U(i+1,j) + U(i-1,j) +
	  U(i,j+1) + U(i,j-1))/5								

2 vs. 1	U, UN are (*,BLOCK); i,j loops INDEPENDENT	0.7	2.8	10.1	26.4	0.3	1.4	4.2	10.8
3 vs. 1	U, UN are (*,BLOCK); with forall	0.7	2.5	10.0	26.4	2.4	8.2	30.5	80.6
4 vs. 1	U, UN are (*,BLOCK); with forall and eoshift	0.5	1.9	7.1	19.0	0.8	2.3	9.7	28.3
5 vs. 1	U, UN (BLOCK,BLOCK); i,j loops INDEPENDENT	0.7	2.4	8.2	20.7	0.4	1.3	5.0	13.3
6 vs. 1	U, UN (BLOCK,BLOCK); with forall	0.7	2.5	8.2	20.7	2.1	6.3	18.7	48.7
7 vs. 1	U, UN (BLOCK,BLOCK); with forall and eoshift	0.7	2.4	8.1	20.5	2.0	6.0	18.2	47.3

Table 8 — 5-point Stencil, Speedup Compared to Serial Version

		SP			T3E		
	number of processors:	**4**	**16**	**64**	**4**	**16**	**64**

1	do k=1,nz
	  do j=1,ny
	  do i=1,nx
	  y(i,j,k)=x(i,k,j)
		0.1	0.5	0.2	0.7	0.5	0.2

2	do k=1,nz
	  do j=1,ny
	  y(:,k,j) =
	  transpose(x(:,j,k))
		0.5	0.2	0.1	0.6	0.3	0.1

3	do i=1,nx
	  y(i,:,:) =
	  transpose(x(i,:,:))
		1.1	0.7	0.3	2.3	1.7	1.0

4	do k=1,nz
	  do j=1,ny
	  do i=1,nx
	  x2(i,j,k)=x(i,j,k)
		0.7	0.6	0.2	0.8	0.5	0.3

5	x2=x	0.6	0.6	0.3	3.4	4.5	6.3
6	REDISTRIBUTE x2(*,BLOCK,*)	0.6	0.6	0.3	3.3	4.6	5.8

Table 9 — Intrinsics, Speedup Compared to Uniprocessor Case

		SP				T3E			
	number of processors:	**1**	**4**	**16**	**64**	**1**	**4**	**16**	**64**

1 - serial version	do k=1,nz
	  do j=1,ny
	  do i=1,nx
	  y(i,j,k)=x(i,k,j)								

2 vs. 1	do k=1,nz
	  do j=1,ny
	  y(:,k,j) =
	  transpose(x(:,j,k))
		19	81	8	5	20	16	11	7

3 vs. 1	do i=1,nx
	  y(i,:,:) =
	  transpose(x(i,:,:))
		28	289	45	41	33	109	115	151

4 - serial version	do k=1,nz
	  do j=1,ny
	  do i=1,nx
	  x2(i,j,k)=x(i,j,k)								

5 vs. 4	x2=x	464	384	465	559	210	940	1797	4894
6 vs. 4	REDISTRIBUTE x2(*,BLOCK,*)	439	367	457	496	204	901	1774	4405

Table 10 — Intrinsics, Speedup Compared to Serial Version

		SP				T3E		
	number of processors:	**4**	**16**	**64**		**4**	**16**	**64**

1	do j=1,n
	  do i=1,n
	  C(i,j) = A(i,j) + B(i,j)
		1.0	0.9	0.9		0.8	0.8	0.8

2	A, B, C are (*,BLOCK); C = A + B	4.2	14.2	58.0		1.4	5.6	21.6

Table 11 — Matrix Addition, Speedup Compared to Uniprocessor Case

		SP				T3E			
	number of processors:	**1**	**4**	**16**	**64**	**1**	**4**	**16**	**64**

1 - serial version	do j=1,n
	  do i=1,n
	  C(i,j) = A(i,j) + B(i,j)								

2 vs. 1	A, B, C are (*,BLOCK); C = A + B	1	5	17	69	6	10	41	158

Table 12 — Matrix Addition, Speedup Compared to Serial Version

		SP			T3E		
	number of processors:	**4**	**16**	**64**	**4**	**16**	**64**
inner product							
1	do i=1,n do j=1,n do k=1,m Y(i,j) = Y(i,j) + A(i,k) * X(k,j)	1.0	1.0	1.0	1.0	1.0	1.0
2	X, Y are (BLOCK,*); A is (*,BLOCK)	10.0	48.2	45.7	4.5	7.3	25.2
3	i and j loops are INDEPENDENT	10.2	50.8	45.1	4.1	16.0	32.3
4	A, Y are (BLOCK,*); X is (*,BLOCK); i, j loops INDEPENDENT	10.1	42.2	24.8	4.0	15.8	35.0
5	A, X, Y distributed (BLOCK ,BLOCK); i, j loops INDEPENDENT	7.0	36.5	60.5	4.0	16.2	42.3
column-oriented							
1	do i=1,n do j=1,m do k=1,n Y(k,i) = Y(k,i) + A(k,j) * X(j,i)	1.0	1.0	1.0	1.0	1.0	1.0
2	A, X, Y are (*,BLOCK)	4.1	13.9	26.5	4.0	13.8	20.7
3	i and k loops are INDEPENDENT	3.6	5.6	2.9	4.1	16.3	58.9
4	A, Y are (*,BLOCK); X is (BLOCK,*); i, k loops INDEPENDENT	3.6	5.5	2.7	4.1	15.6	43.4
5	A, X, Y distributed (BLOCK ,BLOCK); i, k loops INDEPENDENT	3.8	12.2	29.5	4.6	18.1	68.8
row-oriented							
1	do i=1,n do j=1,m do k=1,n Y(i,k) = Y(i,k) + A(i,j) * X(j,k)	1.0	0.8	0.8	1.0	1.0	1.0
2	A, X, Y are (BLOCK,*)	9.0	42.0	25.8	4.5	5.9	1.7
3	i and k loops are INDEPENDENT	4.6	15.7	36.2	5.1	20.5	48.2
4	X, Y are (BLOCK,*); A is (*,BLOCK); i, k loops INDEPENDENT	4.6	15.8	37.4	5.0	20.8	62.5
5	A, X, Y distributed (BLOCK ,BLOCK); i, k loops INDEPENDENT	7.4	551	1361	5.0	21.0	79.8

Table 13 — Matrix Multiply, Speedup Compared to Uniprocessor Case

		SP				T3E			
	number of processors:	**1**	**4**	**16**	**64**	**1**	**4**	**16**	**64**
inner product									
1 - serial version	do i=1,n do j=1,n do k=1,m Y(i,j) = Y(i,j) + A(i,k) * X(k,j)								
2 vs. 1	X, Y are (BLOCK,*); A is (*,BLOCK)	1	9	47	44	2	11	18	62
3 vs. 1	i and j loops are INDEPENDENT	1	10	51	45	1	2	9	19
4 vs. 1	A, Y are (BLOCK,*); X is (*,BLOCK); i, j loops INDEPENDENT	1	10	42	25	1	2	10	21
5 vs. 1	A, X, Y distributed (BLOCK ,BLOCK); i, j loops INDEPENDENT	1	7	37	61	1	2	10	26
column-oriented									
1 - serial version	do i=1,n do j=1,m do k=1,n Y(k,i) = Y(k,i) + A(k,j) * X(j,i)								
2 vs. 1	A, X, Y are (*,BLOCK)	0.1	0.4	1	3	2	10	33	49
3 vs. 1	i and k loops are INDEPENDENT	2	5	9	4	1	3	11	39
4 vs. 1	A, Y are (*,BLOCK); X is (BLOCK,*); i, k loops INDEPENDENT	2	5	8	4	1	3	11	29
5 vs. 1	A, X, Y distributed (BLOCK ,BLOCK); i, k loops INDEPENDENT	2	6	18	45	1	3	11	41
row-oriented									
1 - serial version	do i=1,n do j=1,m do k=1,n Y(i,k) = Y(i,k) + A(i,j) * X(j,k)								
2 vs. 1	A, X, Y are (BLOCK,*)	3	23	130	81	2	11	14	4
3 vs. 1	i and k loops are INDEPENDENT	1	4	16	37	1	2	9	21
4 vs. 1	X, Y are (BLOCK,*); A is (*,BLOCK); i, k loops INDEPENDENT	1	4	16	38	1	2	9	28
5 vs. 1	A, X, Y distributed (BLOCK ,BLOCK); i, k loops INDEPENDENT	1	6	553	1372	1	2	10	36

Table 14 — Matrix Multiply, Speedup Compared to Serial Version

		SP			T3E		
	number of processors:	**4**	**16**	**64**	**4**	**16**	**64**

1	do j=1,n
	  do i=1,n
	  C(i,j)=A(i,j) * B(i,j)
		1.0	0.9	0.9	0.8	0.8	0.8

2	A,B,C are (*,BLOCK); C = A * B	3.9	14.3	59.0	2.8	0.3	43.4

3	do j=1,n
	  do i=1,n
	  C(i,j)=A(i,j) * B(i,j)
	  + D(i,j)
		0.9	0.9	0.9	0.8	0.8	0.8

4	arrays are (*,BLOCK); C = A * B + D	3.8	13.0	51.4	1.5	5.9	1.5

5	do j=1,n
	  do i=1,n
	  C(i,j)=A(i,j) * k
		1.1	1.1	60.3	0.9	0.9	0.9

6	A,C are (*,BLOCK); C = A * k	4.8	17.1	68.4	1.6	7.0	1.6

7	do j=1,n
	  do i=1,n
	  C(i,j)=A(i,j) * k + m
		1.0	1.0	1.0	0.9	0.9	0.9

8	A,C are (*,BLOCK); C = A * k + m	4.4	16.1	52.5	1.7	6.8	1.7

Table 15 — Mult/Add by Matrix Elt, Speedup Compared to Uniprocessor Case

		SP				T3E			
	number of processors:	**1**	**4**	**16**	**64**	**1**	**4**	**16**	**64**

1 - serial version	do j=1,n
	  do i=1,n
	  C(i,j)=A(i,j) * B(i,j)								

2 vs. 1	A, B, C are (*,BLOCK)	1.1	4.2	16.9	70.3	2.9	10.3	1.1	157.8

3 - serial version	do j=1,n
	  do i=1,n
	  C(i,j)=A(i,j) * B(i,j)
	  + D(i,j)								

4 vs. 3	A, B, C are (*,BLOCK)	1.2	4.9	16.6	67.0	5.3	10.1	40.5	10.1

5 - serial version	do j=1,n
	  do i=1,n
	  C(i,j)=A(i,j) * k								

6 vs. 5	A, C are (*,BLOCK)	1.0	4.6	16.3	1.2	6.8	12.6	53.6	12.6

7 - serial version	do j=1,n
	  do i=1,n
	  C(i,j)=A(i,j) * k + m								

8 vs. 7	A, C are (*,BLOCK)	1.0	4.6	16.8	55.2	6.7	12.8	52.5	12.8

Table 16 — Mult/Add by Matrix Element, Speedup Compared to Serial Version

table	test type	# same out of top two	# same out of top three	same order?
Table 1	transpose	0	2	none
	redistribute	1	2	same #1
Table 2	transpose	0	1	none
	redistribute	1	2	same #1
Table 3	scatter	1	2	same #1
	gather	2	3	top 3
	dense scatter	1	3	none
	dense gather	1	3	none
Table 4	scatter	2	3	top 3
	gather	2	3	top 3
	dense scatter	2	–	top 2
	dense gather	2	–	top 2
Table 5	max	1	2	same #1
	sum	1	2	same #1
	sum() with abs	2	3	none
Table 6	max	1	3	none
	sum	2	3	top 3
	sum() with abs	2	3	none
Table 7	stencil	1	2	same #1
Table 8	stencil	1	2	none
Table 9	transpose	1	3	same #1
	REDISTRIBUTE	2	3	none
Table 10	transpose	2	–	top 2
	REDISTRIBUTE	2	–	top 2
Table 13	inner	1	2	same #1
	column	1	2	same #1
	row	2	3	top 3
Table 14	inner	1	2	none
	column	1	2	none
	row	1	2	same #1

Table 17 — # of Loops Ranked in the Top 2 or 3 on Both Machines

tables	test type	SP — same out of top 3	T3E — same out of top 3
Tables 1 & 2	transpose	0	2
	redistribute	2	2
Tables 3 & 4	scatter	2	3
	gather	2	2
	dense scatter	1 of 2	1 of 2
	dense gather	2 of 2	1 of 2
Tables 5 & 6	max	2	2
	sum	3; same order	2
	sum() with abs	2	1
Tables 7 & 8	stencil	2	1
Tables 9 & 10	transpose	2 of 2; same order	2 of 2; same order
	REDISTRIBUTE	2 of 2; same order	2 of 2; same order
Tables 13 & 14	inner	3; same order	2
	column	2; same #1	2
	row	2; same #1	3; same order

Table 18 — # of Loops Ranked Highly for Both Kinds of Speedup